Can Seqlocks Get Along with Programming Language Memory Models?

Hans-J. Boehm  HP Labs

Seqlocks

Read-write locks cause coherence misses on reads.
Even reader critical sections modify lock.
Reader critical sections need exclusive cache-line access.

Seqlocks: Readers don’t update a lock data structure.
Writers increment sequence number before and after change.
Readers check whether writer interfered.

Used in Linux kernel, jsr166e SequenceLock.
Similar techniques used for e.g. software transactional memory implementations.

Seqlocks, version 0 (C++11, broken)

atomic<unsigned long> seq:

int data1, data2;

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
        seq0 = seq;
        r1 = data1.load(m_o_acquire);
        r2 = data2.load(m_o_acquire);
        atomic_thread_fence(m_o_acquire);
        seq = seq.load(m_o_acquire);
    } while (seq0 != seq1 || seq0 & 1);
    do something with r1 and r2;
}

Can be reordered!
“seq_cst” load does not enforce order w.r.t. earlier ops

Seqlocks, version 1 (correct)


c[atomic<] long seq;

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
        seq0 = seq.load(m_o_acquire);
        r1 = data1.load(m_o_acquire);
        r2 = data2.load(m_o_acquire);
        atomic_thread_fence(m_o_acquire);
        seq = seq.load(m_o_acquire);
    } while (seq0 != seq1 || seq0 & 1);
    do something with r1 and r2;
}

Easy, principled fix.
No data races ⇒ sequentially consistent
Prevents data reads from being reordered
Expensive on POWER.
Roughly 3x worse than optimal.
Roughly optimal on x86.
Atomic accesses to data initially surprising, but make sense.

Seqlocks, version 2 (broken)

atomic<unsigned long> seq:

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
        seq0 = seq;
        r1 = data1.load(m_o_acquire);
        r2 = data2.load(m_o_acquire);
        seq = seq.load(m_o_acquire);
    } while (seq0 != seq1 || seq0 & 1);
    do something with r1 and r2;
}

Can be reordered!
“seq_cst” load does not enforce order w.r.t. earlier ops

Seqlocks, version 3 (correct, fast, tricky, C++-only)

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
        seq0 = seq.load(m_o_acquire);
        r1 = data1.load(m_o_acquire);
        r2 = data2.load(m_o_acquire);
        atomic_thread_fence(m_o_acquire);
        seq = seq.load(m_o_acquire);
    } while (seq0 != seq1 || seq0 & 1);
    do something with r1 and r2;
}

Seqlocks, version 4 (correct, scalable with compiler optimization)

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
        seq0 = seq.load(m_o_acquire);
        r1 = data1.load(m_o_acquire);
        r2 = data2.load(m_o_acquire);
        seq = seq.fetch_and_add(0, m_o_release);
    } while (seq0 != seq1 || seq0 & 1);
    do something with r1 and r2;
}

Version 1 (seq. cst. for data) is easy to write, works with C++ and Java, performs well on some platforms, not others.
Version 3 (fences) is very tricky to write correctly. Should perform well everywhere. Only for C++.
Version 4 (read-don’t-modify-write) works everywhere. Scalability depends on currently unimplemented compiler optimization. With optimization: Worse than version 1 on X86, better on POWER.
Version 2 (plain relaxed data) is probably quite popular in Java, but is undeserving of its popularity.

Other languages:
C++11 really uses more verbose memory order syntax, e.g. memory_order_relaxed.
C11 memory model is essentially identical to C++11, modulo syntax.
Java volatile corresponds to atomic<T> with default seq_cst operations. Ordinary data operations behave similarly to C++ m_o_relaxed.
C# memory model is probably close enough to Java for this purpose, though it’s generally considerably different.